

# Towards an industrial strength SQL/XML Infrastructure

Muralidhar Krishnaprasad, Zhen Hua Liu, Anand Manikutty, James W. Warner, Vikas Arora

Oracle Corporation

400. Oracle Parkway

Redwood Shores, CA 94065

USA

{Muralidhar.Krishnaprasad, Zhen.Liu, Anand.Manikutty, Jim.Warner, Vikas.Arora}@oracle.com

## ABSTRACT

XML has become an attractive data processing model for applications. SQL/XML [1] is a SQL standard that integrates XML with SQL. It introduces the XML datatype as a native SQL datatype and defines XML generation functions in the SQL/XML 2003 standard [2]. The goal for the next version of SQL/XML is integrating XQuery [3] with SQL by supporting XQuery embedded inside SQL functions such as the *XMLQuery* and *XMLTable* functions. [5,6,7]. Starting with the 9i database release, Oracle has supported the XML datatype and various operations on XML instances. [9] In this paper, we present the design and implementation strategies of the SQL/XML standard in Oracle XMLDB. We explore the various critical infrastructures needed in the SQL database kernel to support an efficient native XML datatype implementation and the design approaches for efficient generation, query and update of the XML instances. Furthermore, we also illustrate extensions to SQL/XML that makes Oracle XMLDB a truly industrial strength platform for XML processing.

## 1. Introduction

XML has become the de facto standard for data storage, processing and interchange between applications. Having an XML datatype in a SQL system enables applications to use XML as the first-class built-in datatype and use it as datatype for columns of tables and views, and parameters and return values of user defined SQL functions, stored procedures, and triggers. Without the support of XML as a native datatype in a SQL system, mid-ware database applications have to either use a CLOB based datatype to store XML or explicitly shred XML into multiple relational tables and write XML infrastructure logic in their applications to generate, query and update XML. This approach not only results in non-trivial implementation efforts in the application itself, but also having intrinsic performance issue as the underlying relational data may have to be transferred outside the database engine to the application where the actual data processing logic resides.

With native support of the XML datatype, Oracle XMLDB allows users to store XML in a declarative manner, to generate, query and update XML all using the declarative SQL language, leaving application developers to focus on their business logic.

XML can be stored in XMLType table or tables with XMLType columns in Oracle XMLDB. Besides a CLOB storage option, schema [4] based XML can be stored object relationally for efficient query and update capability [10,11].

XML can be generated from arbitrarily complex relational SQL using XML generation functions, such as *XMLElement*, *XMLForest*, and *XMLAgg*, defined by SQL/XML 2003 [2]. This

enables users to generate XML from relational data using SQL. Relational views of XML can also be created on XML stored in XMLType tables. Schema based XML views can also be generated using SQL/XML generation functions.

Querying of XML tables and views is currently supported using XPath embedded inside Oracle provided SQL functions, such as, *Extract*, *ExtractValue*, *ExistsNode* and the *XMLSequence* table function [12]. The on-going SQL/XML effort for the next version of the standard is introducing *XMLQuery*, *XMLCast*, *XMLExists* and *XMLTable* [5,6,7], which are principally the same as the above-mentioned Oracle functions except they allow the embedding of XQuery instead of XPath. We will support them as they become the standard.

While supporting DML over XML is currently not addressed by SQL/XML, Oracle XMLDB supports it via the Oracle extension SQL functions: *updateXML*, *deleteXML*, *insertChildXML*, *insertXMLBefore* and *appendChildXML*.

An industrial strength SQL/XML infrastructure support consists of efficiently implementing XML as a native SQL data type, efficiently generating XML from relational data and efficiently storing, querying and updating XML stored in tables and those generated using XML views over relational data.

On discussing these key components of the infrastructure, the rest of the paper is organized as follows. Section 2 discusses the XML Datatype support. Section 3 discusses the support of XML generation functions. Section 4 discusses support of efficient XML query processing. Section 5 discusses schema based XML. Section 6 discusses support of XML update capability.

## 2. Implementation of XML Data Type

Implementing an efficient XML datatype is critical to the SQL/XML infrastructure. This section talks about the details behind the implementation of the XML datatype in Oracle XMLDB and how the various flexible formats help in speeding up query execution while gracefully scaling for large documents.

### 2.1 Leveraging Type Extensibility in ORDBMS

Supporting a new XML datatype in a relational system requires significant implementation effort, since it is unlike the classical scalar types like *NUMBER* or *VARCHAR*. An object relational enabled database on the other hand, provides the basic type extensibility framework so that new SQL data types can be added into the ORDBMS without significantly modifying the database kernel [13].

We leverage this type extensibility object relational infrastructure available since Oracle 8i to implement the XML datatype as an opaque user defined datatype. Oracle ORDBMS kernel defines a set of interface functions, such as type constructor, destructor, copy, import, export etc for supporting a new opaque user defined type. We provide an implementation for each interface function for the XMLType. This approach allows us to plug in an implementation of XMLType into the Oracle kernel while allowing the database to be agnostic of the internal structure of the type so that all components communicate with each other via well-defined interfaces.

## 2.2 Serialized format of XMLType

In designing the physical serialized form of the opaque XML type, we realize that there is no “one size fits all” solution since the serialized XML value needs to be transported under different scenarios for different consumers of the data. Some common transport scenarios are those between SQL operands and functions, between server components like PL/SQL and Java, between parallel query processes, between client and server network transport via high level JDBC and Oracle OCI APIs, and between remote database instances. Consumers differ in terms of their preference for the format for consumption and may be physically limited to supporting fewer formats. Also, each format is optimal in certain scenarios. Hence multiple serialized representations and conversions between them must be supported for optimal performance. For instance, when transported between two SQL operands within the same row-source, the serialized form can use an internal pointer format, whereas when passing between remote database instances, an inline text format may be used.

To achieve this flexibility and support various storage options, we designed the XML datatype to have multiple serialized forms.

The serialized form consists of version and flag bytes followed by the actual payload. The flag bytes indicate the form of the payload such as LOB locator, inline text data, in-memory pointer etc., type information such as the XMLSchema associated with the document, processing information such as whether the document is valid or well formed etc., and other auxiliary information such as whether spaces should be preserved when parsing the document.

Ver	Flags	SchemaId	Element#	Payload
V1	LOB, VALID			-----LOB Locator-----
V1	INLINE			“<PO id=“3”>...</PO>”
V1	POINTER			-- in memory pointer ----

Figure 1 – XML datatype formats

Figure 1 shows the format of the datatype and some sample payloads. Furthermore, this design of the XML datatype is flexible enough that new payload formats can be added relatively easily. This architecture also helps preserve compatibility with older clients, as the payload in a newer format can be converted to the format understood by the client at the server boundaries.

## 2.3 Flexible pointer format

To illustrate the advantage of having multiple formats for the serialized representation, assume we have a table “potab” having an XMLType column “po” which uses CLOB storage. Now consider the query in Table 1 that extracts the purchase order number, the shipping address street, city and state, using XPath expressions embedded inside the Extract function.

Extract operations can be complex navigational operations, and are often performed by manifesting the XML as a DOM tree for CLOB based storage. A traditional, naïve implementation might create four DOM trees for the four consumers of data - the four extract operators. Obviously this method is quite inefficient since the DOM tree has to be instantiated for each operator.

```
SELECT Extract(v.po, '@pono') as "pono",
       Extract(v.po, '/ShipAddr/Street') as "street",
       Extract(v.po, '/ShipAddr/City') as "city",
       Extract(v.po, '/ShipAddr/State') as "state"
FROM potab v;
```

Table 1- Multiple Extract functions

With the flexible serial format, one of the payload formats can be a physical memory pointer. Because of this flexibility, during query evaluation, for each row, the XML image for the po column can be created by first instantiating an in-memory tree corresponding to the XML value and then stashing it's pointer in the image. This image is passed to the various extract operators, which can access the tree directly without having to instantiate it first. Thus, the various consumers of data can share the in-memory data structures without the need to create multiple copies. The result of the extract can also be serialized in a similar fashion for input to further consumers down the pipeline.

The figure below shows a sample of this optimization with a query involving multiple extract operations over various sizes of XML documents. Opt (n) indicates the optimized evaluation with n extracts and Trad (n) indicates the same with the traditional evaluation. The graph is normalized against the time taken for the Opt (5) case.

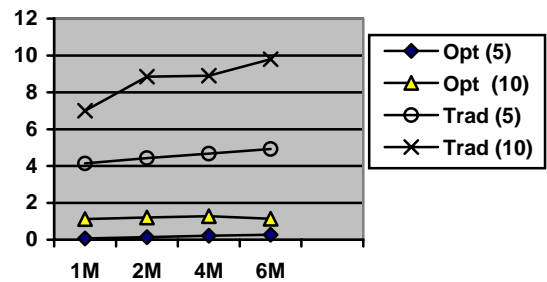


Figure 2 – Performance of optimized and traditional formats

At the server boundary, there are scanning routines that scan the XML image and convert the payload in to a network neutral format that the client can understand such as inline text data or CLOBs.

## 2.4 Reference count and Garbage collection of XMLType objects

The traditional relational system simply works on scalar values that are copied through the various operators and row sources. Hence there is no need to garbage collect any data values. However, in the case of XMLType pointer formats described above we need to maintain reference counts, since we have created in-memory trees and we need to clean them up. This is done by having the in-memory DOM tree contains a reference counter that is incremented by a special garbage collection routine that scans the XMLType image as it is copied across the SQL row sources and operators. During the next fetch of the SQL row source, the garbage collector decrements the reference count of the XMLType. When the reference count goes to zero then the XMLType DOM tree is freed and memory recovered. This garbage collection mechanism ensures that memory for the DOM trees are used efficiently.

## 3. XML Generation from Relational Data

Along with an efficient XML datatype implementation the SQL/XML infrastructure must allow for a fast and scalable XML generation. Generating XML from relational data is a very typical requirement in most XML applications as there is a vast amount of data stored in relational systems. Generating XML from relational data using SQL/XML generation functions and creating XML view over SQL using generation functions enables users to create virtual XML by leveraging their existing relational data.

### 3.1 SQL/XML Functions for generation

Consider a classical relational example where there are tables: *depts*, *emps*, *schools*, *kids*, *equipments* forming master-detail relationships. Each department has multiple employees and multiple equipments. Each employee has multiple kids and has attended multiple schools.

Table 2 reflects this master detail relationship in XML. It shows a department XML document instance that consists of a collection of "Employee" element nodes and a collection of "Equipment" element nodes under the top "Dept" element node. There are a collection of "kid" element nodes and "school" element nodes under each "Employee" element node.

To generate multiple rows of department XML document instances, we can use the SQL/XML generation functions in a SQL statement as shown in Table 3. Note that *XMLAgg* is used with a correlated subquery to form an aggregate of collection elements. The nested *XMLAgg* invocation reflects the nested collection elements in the XML documents.

```
<Dept>
  <DeptInfo DeptNo="10">
    <DeptName>Accounting</DeptName>
    <Location>Building 300</Location>
  </DeptInfo>
  <Employee>
    <Ename>Smith</Ename>
    <Job>VP</Job>
    <kid>
      <KidName>Peter</KidName>
    </kid>
    <school>
      <SchoolName>CMU</SchoolName>
```

```
</school>
  <school>
    <SchoolName>UCSD</SchoolName>
  </school>
</Employee>
<Employee>
  <Ename>Clark</Ename>
  <Job>Manager</Job>
  <kid>
    <KidName>Grace</KidName>
  </kid>
  <kid>
    <KidName>Mark</KidName>
  </kid>
</school>
  <SchoolName>UCB</SchoolName>
</school>
</Employee>
<Equipment eid="1">
  <Eqpname>Auditor tool</Eqpname>
</Equipment>
<Equipment eid="2">
  <Eqpname>financial caculator</Eqpname>
</Equipment>
</Dept>
```

Table 2 - One row of department XML Document

```
SELECT
XMLElement("Dept", XMLAttributes(deptno as "DeptNo"),
XMLElement("DeptInfo",
XMLForest(dname as "DeptName", loc as "Location")),
XMLConcat(
(SELECT XMLAgg(XMLElement("Employee",
XMLForest(ename as "Ename", job as "Job"),
(SELECT XMLAgg(
XMLElement("kid", XMLForest(name as "KidName")))
FROM kids
WHERE kids.empno = emps.empno),
(SELECT XMLAgg( XMLElement("school",
XMLForest(name as "SchoolName")))
FROM schools
WHERE schools.empno = emps.empno )
FROM emps
WHERE emps.deptno = d.deptn ),
(SELECT XMLAgg(XMLElement("Equipment",
XMLAttributes(equipid as "eid"),
XMLForest(equipname as "Eqpname"))))
FROM equipments
WHERE equipments.equipDptid = d.deptno)))
FROM depts d
```

Table 3 - SQL constructing department XML documents

### 3.2 Top-down Stream Generation

Efficiently generating XML documents from relational data has been researched in the past. [14,15]. For SQL statements with SQL/XML generation functions, such as the XML generation SQL shown in Table 3, Shanmugasudaram et al. [14] concludes that the idea of "late tagging and early structuring" with "sorted, outer-union" approach of generating XML inside relational engine

provides an efficient and robust way to generate XML from relational data.

We approach the generation problem in a different way. Rather than implementing outer-join natively, we use the strategy of minimizing CLOB generations via top-down streaming evaluation of SQL/XML generation functions.

The nested nature of SQL/XML generation functions lends itself naturally to the traditional bottom-up evaluation strategy. That is, each level of the SQL/XML generation function within the entire expression tree can be evaluated independent of its ancestors in the tree. However, such a strategy is sub-optimal since the data at the lowest layers is continually copied up the expression tree. The operation would be much more efficient if all SQL/XML generation functions could share the same data destination.

This goal is accomplished by evaluating the SQL/XML generation function expression tree in a top-down manner. When evaluating the generation expression tree this way, the topmost SQL/XML generation function creates a stream over the desired destination for capturing the XML output. This stream is then passed down to the children in a top-down and depth-first manner, with each descendant writing its XML generation result to it directly.

The stream abstraction is an API to write and read bytes. The physical implementation of the may use a contiguous buffer, segmented array memory, CLOBs or network buffers. By using this stream abstraction, we can choose the most optimal stream implementation for a specific task, while the rest of the engine works over the abstraction. Section 3.3 talks about the physical stream mappings in detail.

### 3.2.1 Top Down Evaluation

Consider the simple SQL/XML query shown in Table 4:

```
SELECT XMLElement("Dept",
    XMLAttributes(deptno as "DeptNo"),
    XMLElement("DeptInfo",
        XMLForest(dname as "DeptName", loc as "Location")))
FROM depts. d
```

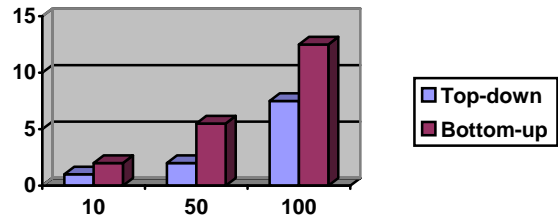
**Table 4 - Simple XML generation SQL**

With the naïve approach of doing bottom up evaluation, the *dname* column data, for example, is copied first into the *XMLForest* function, then to the "DeptInfo" *XMLElement* function. It is then copied again into the top "Dept" *XMLElement*, for a total of three copies of the same data.

With the top-down approach, however, the top "Dept" *XMLElement* function creates a stream. It prints the opening tag for "Dept" into the destination stream, and then passes this stream to *XMLAttributes*. When this is finished, it passes the same stream down to the child *XMLElement* "DeptInfo" function, who in turn passes the stream to the *XMLForest*. After "DeptInfo" *XMLElement* function returns, it prints the closing tag for "Dept". In this way, each *datum* is copied only once into the stream.

This approach solves one of the key performance problems resulting from the Correlated CLOB approach described by [14]. In this top down streaming approach, we don't generate large number of CLOBs to hold the intermediate results of XML generation. The costly repeated creation and copying of CLOBs with each invocation of XML generation function is completely avoided.

Figure 3 compares the performance of the top-down approach to the naïve bottom-up approach. The X-axis indicates the depth of the SQL/XML tree while the Y-axis indicates the time taken to perform the generation normalized against the *depth(10)* case. As can be seen, the top-down approach performs better than bottom-up approach consistently.



**Figure 3 - SQL/XML Generation Function Top-down Vs Bottom-up Evaluation Performance**

### 3.2.2 XMLAgg optimization

This top-down stream evaluation strategy can also be applied to *XMLAgg* in correlated subquery SQL statement as shown in Table 3 as well. Handling streaming optimization for an aggregate function like *XMLAgg* is more involved, as the conceptual bottom up approach evaluation strategy is that the aggregation is done after the evaluation of the inner SQL/XML operands. The inner SQL/XML operands can be stream evaluated in a top down fashion, but their results need to be aggregated and copied again. To avoid this unnecessary copy, we perform two kinds of optimization depending on whether there requires sorting of the relational data before XML is formed as the streaming operation naturally works better for non-blocking operations.

#### 3.2.2.1 Shared Stream for non grouped case

In the case when the *XMLAgg* is not associated with a *GROUP BY* clause and the *XMLAgg* does not contain an *ORDER BY* clause, the top down streaming evaluation still applies. With the top down approach evaluation of the query in Table 3, the key is to have each *XMLAgg* function write its data directly to the stream created by the top "Dept" *XMLElement*. This is conceptually accomplished by each parent SQL/XML generation function passing down the stream to its children.

The logical passing down stream at run time among SQL/XML generation functions is accomplished by having all the generation function fetch the destination stream from a common location. During query compile time, we stash an address of a stream location in the meta-data of each SQL/XML generation function in the entire expression tree. Then during run time, the top level SQL/XML generation function creates the destination stream and stores the reference to the stream in that location so that each SQL/XML generation function can write into the result stream whose location is stashed in its meta-data when it is called for evaluation.

#### 3.2.2.2 Group by/Order by Optimization

When the subquery with the *XMLAgg* contains a *GROUP BY* or the *XMLAgg* function contains an *ORDER BY*, the top down optimization breaks down with a traditional evaluation. This is due to the fact that the operands need to be evaluated and ordered

before aggregation. Consider the query shown in table 5 that groups all employees according to their department.

To optimize this case, we delay the evaluation of the SQL/XML operand tree until after the sorting or hashing of the leaf scalar values are performed. In this example, the *deptno*, *ename* and *empno* columns are grouped first. The *XMLAgg* is then called for each distinct group which sets up the stream and the SQL/XML operands are then evaluated using that stream.

```
SELECT XMLAgg(
    XMLElement ("emp", XMLForest(ename, empno))
from emp
group by deptno;
```

**Table 5- Example XMLAgg with grouping**

### 3.2.2.3 Analysis of the top down approach

The disadvantage of the approach described here is that the query has to be evaluated as it is written, correlated subquery has to be evaluated in nested-loop fashion and query de-correlation by relational optimizer cannot be leveraged. The strength of "outer-union" approach is that it generates one outer-union query and gives the optimizer much more flexibility to optimize the entire query.

However, Fernandez et al. [15] compares the "sorted outer-union" approach using one query with its naive "fully partitioned" strategy with multiple SQL queries that do not contain outer joins or outer unions and concludes that the neither approach is optimal and that the optimal approach generates multiple SQL queries, which may contain outer join or outer unions.

We choose to use our "Top down streaming" approach since enhancing the relational optimizer to optimize a giant SQL query with many left outer-join and outer-union is a non-trivial exercise. Furthermore, the "Top down streaming" approach resolves the intermediate CLOB generations, construction and copy issues, which are the key performance bottlenecks for XML generation. The "sorted outer union" approach increases the response time since it requires sorting even though the original query may not have a *GROUP BY* or *ORDER BY* clause in the *XMLAgg*, whereas the "Top down streaming" approach can stream the XML data out immediately.

Other than the "Top down streaming" approach, the XML generation performance can be further improved by intelligently selecting the physical mapping of the logical stream and scalable CLOB technique discussed below.

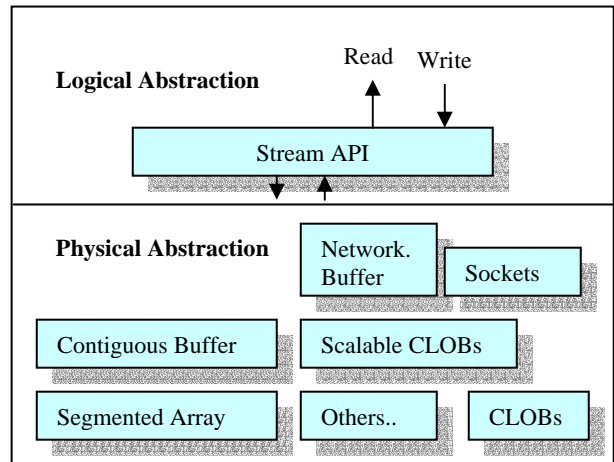
### 3.3 Stream physical mapping

The result stream that every SQL/XML generation function writes into is a logical stream that can be created by choosing the most efficient physical stream based on how the result XML is consumed. If the result of the query is directly sent to a database client, which is the most common case, then this logical stream can be created on top of the underlying network buffer stream sent to the client. This enables streaming the results to the client directly as there is no physical CLOB creation at all. If the result of the query is inserted into an XMLType table, as the query is part of the INSERT/SELECT statement, the logical stream can be created on the data buffer stream for the underlying target table. If the result is consumed by an XML schema validation process, for

example, the logical stream can be created on top of the read stream used by the schema validator.

The result of the query, however, can always go into a temporary on disk CLOB. The biggest advantage of using a CLOB is that it scales to include large amounts of data, as an XMLType value is unbounded. However, there is a significant amount of overhead with CLOB, and not all XML documents generated with SQL/XML generation functions are large enough to warrant being stored in a CLOB.

Figure 4 illustrates the stream abstraction.

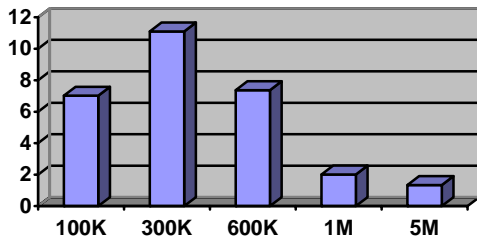


**Figure 4- Stream Abstraction**

### 3.4 Scalable LOBs

In order to handle the small sized XML cases efficiently and scale for the large cases gracefully, SQL/XML functions use a new type of CLOB, called *scalable CLOB*, for their operations. The scalable CLOB is different from the disk-based CLOB as the scalable CLOB keeps their data in memory, only going to disk if the CLOB gets large. This "spill-over" mechanism provides the scalability of CLOB without the expense of going to disk for small size XML documents. This is especially important for applications such as SQL/XML generation function, since the data is generated from the query and may never be stored.

Figure 5 demonstrates how using scalable CLOB with the spillover mechanism can improve performance significantly to generate small size XML data. The X-axis indicates the size of the generated XML document; the Y-axis shows the time taken with disk-based CLOB divided by the time taken with scalable CLOB to generate that size XML document. As can be seen, scalable CLOB improve performance significantly until the spillover point is reached (which is set to 300K for this case). When the generated XML data is 300K, the disk based CLOB is more than 10 times slower than the scalable CLOB. However, when the generated XML document size gets quite large, there is very little advantage for scalable CLOB as the generation time ratio closes to 1 as expected.



**Figure 5 – XML Generation Time Ratio of Disk-based CLOB Vs Scalable CLOB.**

### 3.5 Recursive XML generation

One important aspect of XML is that it allows recursive structures. SQL99 defines the construct of recursive SQL using the WITH clause construct which is an ideal construct to generate recursive XML. An Oracle extension of SQL supports recursive SQL queries using the *CONNECT BY* construct. However, current SQL/XML has not addressed the issue of generating recursive XML documents based on a recursive SQL construct. Oracle XMLDB supports the mapping of a *CONNECT BY* query result into a recursive XML document via the DBMS\_XMLGEN package [12]. We believe that this might be addressed in a future standard or as a vendor extension.

## 4. Efficient XML Querying

XML generation can be achieved using the SQL/XML generation functions as described or by serializing the relational data and late tagging them in the application tier. However, using SQL/XML functions for generation in the database also enables a major functionality that database is meant to perform efficiently – querying of the generated XML instances.

### 4.1 Querying Functions in Oracle

Oracle provides a number of querying functions inside SQL to query XML data. Since XML is tree structured, Oracle introduced a SQL *Extract* function, which extracts XML nodes out of the tree using XPath. If the result of the extraction is a singleton node, such as an element node of simple content, or an attribute node or a text node etc, then the simple content of the node can be further extracted and converted into a simple SQL scalar value. This is accomplished by the *ExtractValue* function. On the other hand, if the result is a forest of element nodes, we should be able to convert it into a virtual relational table so that each node in the forest becomes a row in a virtual table. To accomplish this, Oracle supports the *XMLSequence* table function. [12]

For example, consider a SQL/XML view, shown in Table 6, which is a view of the SQL query in Table 3:

```
CREATE VIEW depts_xml_view (dept_xml) AS
/* SQL/XML Query shown in table 3*/
```

**Table 6 - dept\_xml SQL/XML View**

The SQL statement shown in table 7 queries the *dept\_xml\_view* SQL/XML view using *extract*, *extractValue*, *existsNode*, table of *XMLSequence* functions and returns results shown in table 8.

The construct *XMLSequence* of *Extract* used in table function, as shown in table 7, is a common construct that converts a collection of element nodes in the XML document into a virtual relational table that can be queried relationally. In this case, the *Extract* returns forest of XML element nodes; the *XMLSequence* function breaks down the forest into a SQL collection of XMLType instances each containing a single element node. Then applying the table function converts the SQL collection into a virtual relational table with each row containing an XMLType instance. The SQL statement is a left correlated join and for each row from the *depts\_xml\_view*, a new set of correlated rows is generated from the table function.

Such constructs can be chained into multiple correlated TABLE functions containing the *XMLSequence* function and the *Extract* function to un-nest hierarchical information contained in the XML document. The SQL statement in Table 7 shows a collection of *Employee* element nodes under each *dept* node and a collection of *school* element nodes under each *Employee* element node is converted into virtual relation tables *v\_emps* and *v\_schools*.

The *ExistsNode* SQL function is typically used in the SQL WHERE clause and it tests the existence of the nodes by applying the XPath on the input XMLType instance and returning 1 if there are any nodes qualified by the XPath.

```
SELECT
  Extract (Value(v_emps), '/Employee/Ename') as "EnameXML",
  ExtractValue(Value(v_emps), '/Employee/Job') as "job",
  ExtractValue(Value(v_schools), '/school/SchoolName')
    as "schoolname"
FROM depts_xml_view,
  TABLE(XMLSequence(
    Extract(depts_xml_view.dept_xml, '/Dept/Employee'))
    v_emps,
  TABLE(XMLSequence(EXTRACT(Value(v_emps),
'/Employee/school'))) v_schools
WHERE ExistsNode(dp.dept_xml, '/Dept[@DeptNo=10]') = 1
```

**Table 7 - SQL/XML Query**

EnameXML	job	schoolname
<Ename>Smith</Ename>	VP	UCSD
<Ename>Smith</Ename>	VP	CMU
<Ename>Clark</Ename>	Manager	UCB

**Table 8 - SQL/XML Query Result**

### 4.2 Querying Functions in next release of the SQL/XML standard

The current proposal for the next version of SQL/XML introduces the *XMLQuery*, *XMLExists*, and *XMLCast* SQL functions and XMLTable constructs.[5,6,7] to query XML in SQL. These functions are conceptually the same as Oracle's extension SQL functions: *Extract*, *ExistsNode*, *ExtractValue* and *XMLSequence* table function respectively, see Table 9.

To show the equivalency, we can express the SQL/XML query shown in Table 7 as a query using *XMLTable* and *XMLExists* shown in Table 10.

However, the proposed SQL/XML query functions goes one step further and supports the embedding of XQuery, which is a super

set of XPath, pending XQuery becoming a recommendation. Furthermore, the *XMLQuery* input and output can be value based or reference based of XML and the output option of returning content or sequence [5].

Oracle Extension Function	Proposed Equivalent SQL/XML standard function
EXTRACT	XMLQUERY
EXISTNODES	XMLEXISTS
EXTRACTVALUE	XMLCAST
XMLSequence Table function	XMLTABLE

**Table 9– XML Query SQL Functions**

Enabling XQuery inside *XMLQuery* also gives an alternative way to construct XML node and is more XQuery centric. However, combined with the SQL/XML generation functions, using XPath embedded SQL functions in SQL can effectively accomplish a large portion of data driven XML query use cases.

```
SELECT v_emps."EnameXML", v_emps."job",
       v_schools."schoolname"
FROM depts_xml_view d,
     XMLTable('/Dept/Employee' PASSING d.dept_xml
              COLUMNS
                "EnameXML" XML    PATH 'Ename',
                "job"      varchar(9) PATH 'Job',
                "schools"  XML    PATH 'school'
              ) v_emps,
     XMLTable('/school' PASSING v_emps."schools"
              COLUMNS
                "schoolname" varchar(10) PATH 'SchoolName'
              ) v_schools
WHERE XMLEXists(depts_xml_view.dept_xml,
               '/Dept[@DeptNo=10]')
```

**Table 10 - Using XMLTable, XMLEXists for queries in table 6**

### 4.3 Querying XML Techniques

Querying XML constructed from relational data has been studied and researched [15,16,17,18]. After XML has been shredded into the underlying relational storage, the main idea is to efficiently query the XML by translating the XML query into SQL, which queries the underlying data without unnecessarily materializing the XML.

Querying XML constructed from relational data via SQL/XML generation functions using XPath embedded inside *Extract*, *ExistsNode*, *ExtractValue* functions and *XMLSequence* table function can be optimized through the query rewrite technique inside the database engine [11].

The key observation is that the operation of construction of XML via SQL/XML generation functions and the operation of navigation of the XPath to query XML are the inverse of each other and thus the two operations can be canceled. This can be accomplished by developing a mapping algebra for SQL/XML generation functions and XPath navigation functions and then applying algebraic optimizations during query compile time. Using this algebraic approach, Oracle XMLDB is able to rewrite XPath queries over XML into relational queries that can be optimized by classical relational optimizer yielding XML query

performance comparable to its relational equivalent. The query performance achieved via query rewrite is orders of magnitude faster than the naïve approach of materialization of the XML followed by applying an XPath engine on the resultant XML to compute the query result as the intermediate XML fragment generations are not necessary and XPath predicates becomes SQL predicates that facilitate index usage.

Table 11 shows the rewritten relational query after applying the query rewrite technique for query in Table 7. Note all the *Extract*, *ExtractValue*, *ExistsNode* and *XMLSequence* table functions are not present in the final rewritten query and the rewritten query can be optimized by the classical relational optimizer to pickup appropriate index on the underlying tables and choose the best join order of the tables.

```
SELECT XMLElement("Ename", ename) as "enameXML",
       emps.job as "job", schools.name as "schoolname"
FROM schools, depts, emps
WHERE depts.deptno = emps.deptno and
      emps.empno = schools.empno and
      schools.name = 'CMU' and depts..deptno = 10
```

**Table 11 - Rewritten Relation Query for SQL/XML Query in Table 7**

As discussed in the previous section, XML generation inside the server gives better performance than doing it on the client. Being able to efficiently query the XML generated inside server is another powerful reason why XML generation should be done inside server.

## 5. Schema-based XML Support

### 5.1 Schema Validation

SQL/XML introduces the concept of XML schema validation in SQL via the "IS VALID" predicate [8], which addresses the issue of how XML Schema can be registered into a SQL system and how XML Type instances can be validated in SQL via "IS VALID" predicate used in a SQL WHERE clause or a check constraint. Oracle supports importing XML Schema into Oracle XMLDB via the *DBMS\_XMLSCHEMA.registerSchema* function and does schema validation of XML using the *XMLIsValid* SQL function [12]. *XMLIsValid* function supports both the concept of instance data driven validation and validation according to specific XML schemas registered to the Oracle XMLDB illustrated in [8]. *XMLIsValid* checks if the input XMLType instances conform to the specified XML Schema. If an XML schema URL is not specified as a parameter to *XMLIsValid*, and the XML document is schema-based, then conformance is checked against the XMLType instance's own schema.

### 5.2 Storage of schema based XML

After an XML schema is registered into Oracle XMLDB, an XMLType table storing all XML document instances validated according to a specific schema can be created as the following example in Table 12.

```
CREATE TABLE potab of XMLType
XMLSchema 'http://www.po.com/po.xsd'
Element "PurchaseOrder";
```

**Table 12 - Creation of Schema Based XMLType Table**

The XML instances stored in *potab* are automatically shredded and stored into the underlying object relational tables [10]. Query of XML using XPath embedded in *Extract*, *ExistsNode*, *ExtractValue* functions and *XMLSequence* table function, as discussed previously, are rewritten into object relational queries over the underlying OR storage tables [11]. This results in superior query performance to the same query over the XML stored as CLOB. Feng Tian etc [19] shows that shredding structured XML into relational storage give better performance for many XML queries than shredding without taking advantage of the structural information.

When XMLType data is stored as a CLOB, queries involving *ExistsNode*, *Extract* and *ExtractValue* operations, can be speed up by creating function-based indexes. The optimizer can leverage the functional index to avoid costly XPath evaluation during query execution time.

### 5.3 Schema based XMLType View

By leveraging SQL/XML generation functions, Oracle XMLDB supports the Oracle extension of creation of schema based XMLType views as shown in Table 13.

Conceptually, the *poview* view is achieved by first constructing the XML instance from relational table *purchaseorder* and *lineitems* using SQL/XML generation functions followed by XML schema validation against registered *purchaseOrder* schema.

Querying schema based XML view can be efficiently done via query rewrite technique as discussed previously. The additional step for schema based XMLType view is that since the resultant XML is schema based, a validation step must be done in the end. However, since the validation schema is known at compile time, validation step can be optimized away via compile time schema validation.

```
CREATE VIEW poview OF XMLType
XMLSCHEMA http://www.po.com/po.xsd
ELEMENT "PurchaseOrder"
AS SELECT XMLElement("PurchaseOrder",
XMLAttributes( pono as "pono",
'http://www.po.com/po.xsd' AS "xmlns",
'http://www.w3.org/2001/XMLSchema-instance'
AS "xmlns:xsi",
'http://www.po.com/po.xsd'
http://www.po.com/po.xsd'
AS "xsi:schemaLocation"),
XMLElement("ShipAddr",
XMLForest(street as "Street",
city as "City",
state as "State")),
(SELECT XMLAGG(
XMLElement("LineItem",
XMLAttributes(lino as "Itemno"),
XMLForest(liname as "Liname")))
FROM lineitems l
WHERE l.pono = p.pono)) as po
FROM purchaseorder p
```

**Table 13 - Schema based SQL/XML View**

Compile time validation based on the SQL/XML generation expression tree is done to accomplish the following goals:

- Find XML generation functions generating schema elements that are not invalid based on the schema definition;
- Annotate the SQL/XML generation expression tree for supplying construction of default elements specified by the XML schema;
- Annotate the SQL expression tree to supply proper SQL type casting to cast SQL data to the right data type required by the XML schema.

## 6. DML Support on XML

### 6.1 XML Modification SQL Functions

XML modification has not been addressed by the SQL/XML standard yet. However, being able to manipulate the XML in a piecewise way is critical to ensure the performance of any XML database. The critical operations are to update the value of nodes, add new nodes, and delete existing nodes. In particular, operations on collection elements, such as appending to a collection, inserting or deleting nodes from a collection are important. Oracle XMLDB has added Oracle extension SQL functions to allow piecewise operations on XML data. These functions take in a source XML instance, perform the data manipulation operation, and return a new XML instance. The newly returned XML instance may be used in the target of a SQL update statement to modify the underlying stored XML values. These SQL functions are *updateXML*, *deleteXML*, *insertChildXML*, *insertXMLBefore* and *appendChildXML*.

**InsertChildXML** - Oracle extension SQL function that allows inserting new XML nodes into parent element nodes identified by the XPath. The first example in Table 14 shows a SQL update statement that inserts a new *LineItem* element node to a *purchaseorder* XML document.

<p><b>1. InsertChildXML DML Statement</b></p> <pre>UPDATE potab x SET value(x) = insertChildXML (value(x), '/PurchaseOrder/LineItems/LineItem', XMLParse (content '&lt;LineItem Itemno="100"&gt;SQL/XML Discussion &lt;/LineItem&gt;') WHERE ExistsNode(value(x), '/PurchaseOrder[@pono=10'] = 1</pre>
<p><b>2. UpdateXML DML Statement</b></p> <pre>UPDATE potab x SET value(x) = UpdateXML(value(x), '/PurchaseOrder/ShipAddr/State/text()', 'AZ') WHERE ExistsNode(value(x), '/PurchaseOrder[@pono=10'] = 1</pre>
<p><b>3. AppendChildXML DML Statement</b></p> <pre>UPDATE potab x SET value(x) = AppendChildXML (value(x), '/PurchaseOrder', XMLParse(content '&lt;ShipAddress&gt; &lt;Street&gt;Oakr&lt;/Street&gt; &lt;City&gt;Berkeley&lt;/City&gt; &lt;State&gt;CA&lt;/State&gt; &lt;/ShipAddress&gt;') WHERE ExistsNode(value(x), '/PurchaseOrder[@pono=10']= 1</pre>

#### 4. DeleteXML DML Statement

```
UPDATE potab x
SET value(x) = DeleteXML (value(x),
'/PurchaseOrder/LineItems/LineItem[3]')
WHERE ExistsNode(value(x), '/PurchaseOrder[@pono=10'] = 1
```

#### 5. InsertXMLBefore DML Statement

```
UPDATE potab x
SET value(x) = insertXMLBefore (value(x),
'/PurchaseOrder/ShipAddress/City',
XMLParse(content '<AptNum>34</AptNum>'))
WHERE ExistsNode(value(x), '/PurchaseOrder[@pono=10'] = 1
```

**Table 14 - XML DML Statements**

**UpdateXML** - Oracle extension SQL function that allows updates of the text node value of an element node. The second example in Table 14 illustrates updating the *State* element of a *purchaseorder* XML document

**AppendChildXML** - Oracle extension SQL function that appends the given XML nodes as children of the parent nodes identified by the XPath expression from the source XML instance. The third example in Table 14 shows a SQL update statement that appends the *ShipAddress* to the purchase order. The difference between *appendChildXML* and *insertChildXML* is that the latter inserts the new node automatically into the correct position in the parent element based on the schema information.

**DeleteXML** - Oracle extension SQL function that deletes the nodes identified by the XPath expression from the XML instance. The fourth example in Table 14 shows an update statement that deletes the 3rd line item from the purchase order XML document.

**InsertXMLBefore** - Oracle extension SQL function that allows inserting new XML nodes into parent element nodes before the nodes identified by the XPath. The last example in Table 14 shows a SQL update statement that inserts a new *Aptnum* element node before the *City* element node.

These functions combined with the SQL update statement provide great flexibility in updating an existing XML document. Note that these functions can be used as expressions in SELECT statements as well to create new updated transient XML instances.

## 6.2 Rewrites with DML statements

When XML is stored using a CLOB, the only reasonable update is to update the entire document. When XML is stored object relationally, however, we can directly update the underlying relational columns to do the piecewise operation. The query rewrite techniques explained earlier are also applicable to updates. For the rewrite to occur, the source and the target must be the same XMLType column. The 3<sup>rd</sup> update statement in Table 14, for instance, can be rewritten to update the underlying relational storage column:

```
UPDATE potab x
SET x.xmldata."ShipAddr"."State" = 'CA'
WHERE x.xmldata."@Pono" = 1;
```

Similarly collection operations such as inserting or deleting from a collection can be rewritten to act on the underlying collection table directly.

## 7. CONCLUSION

As XML becomes material technology for data engineering and application, SQL based relational and object relational database system have been critical platforms to support XML. The SQL/XML standard makes the marriage of SQL and XML by bridging the gap between SQL and XML. Although the effort of SQL/XML standard is still on-going, the power and flexibility of SQL combined with standard XML data type and XML generation, XML query, XML modification and validation SQL functions is clear and valuable and together they provide a comprehensive SQL language for XML processing in relational and object relational DBMS. Our experience of supporting SQL/XML in Oracle XML DB has shown that it is feasible to come up with an efficient industrial strength implementation of the SQL/XML standard. The key infrastructure consists of efficient support of XML as native SQL data type, efficient generation of XML from relational data, efficiently store, query and update XML document and XML view over relational data. These help in providing a fast and scalable relational platform for XML processing while reusing the years of work that has gone in to the relational model.

## 8. ACKNOWLEDGMENTS

We gratefully acknowledge the contributions of all the members of the Oracle XMLDB development and product management teams. We would especially like to thank Vishu Krishnamurthy and Susan Kotsovolos, for their great managerial support of the SQL/XML project in Oracle XMLDB.

## 9. REFERENCES

- [1] The International Committee for Information Technology Standard H2.3 Task Group, See <http://www.sqlx.org>
- [2] SQL/XML 2003, the first edition of the SQL/XML standard published by the ISO as part 14 of the SQL standard: ISO/IEC 9075-14:2003.
- [3] World Wide Web Consortium, "XQuery 1.0: An XML Query Language", W3C Working Draft, November 2003
- [4] World Wide Web Consortium, "XML Schema Standard", see <http://www.w3.org/XML/Schema>
- [5] Fred Zemke for the H2 ad hoc subcommittee on SQL/XML, "XML Query", ISO/IEC JTC1/SC32 WG3:SIA-042 ANSI NCITS H2 2004-021rl
- [6] Fred Zemke for the H2 ad hoc subcommittee on SQL/XML, "XMLCast", ISO/IEC JTC1/SC32 WG3:SIA-041 ANSI NCITS H2 2004-020rl
- [7] Fred Zemke etc, "XMLTable", ISO/IEC JTC1/SC32 WG3:SIA-043 ANSI NCITS H2 2004-039
- [8] Fred Zemke for the H2 ad hoc subcommittee on SQL/XML, "Is VALID predicate", ISO/IEC JTC1/SC32 WG3:HBA-034 ANSI NCITS H2 2003-343
- [9] Sandeepan Banerjee, Vishu Krishnamurthy, Muralidhar Krishnaprasad, Ravi Murthy: "Oracle 8i - The XML Enabled Data Management System", ICDE 2000.
- [10] Ravi Murthy, Sandeepan Banerjee: "XML Schemas in Oracle XMLDB", VLDB 2003.

- [11] Muralidhar Krishnaprasad, Zhen Hua Liu, Anand Manikuttu, Jim Warner, Vikas Arora, Susan Kotsovolos: "Query rewrite for XML in Oracle XMLDB", VLDB 2004.
- [12] Oracle XMLDB Developers's Guide, Oracle 10g. See <http://otn.oracle.com/tech/xml/xmlldb>
- [13] M. Stonebraker, D. Moore: "Object-Relational DBMSs: The Next Great Wave" Morgan Kaufmann 1996
- [14] Jayavel Shanmugasundaram, Eugene Shekita, Rimón Barr, Michael Carey, Bruce Lindsay, Hamid Pirahesh, Berthold Reinwald: "Efficiently Publishing Relational Data as XML Documents", VLDB2000
- [15] Mary Fernandez, Atsuyuki Morishima, Dan Suciu: "Efficient Evaluation of XML Middle-ware Queries", SIGMOD May 2001
- [16] M. Fernandez, Y. Kadiyska, A. Morishima, D. Suciu, W-C Tan SilkRoute: "a framework for publishing relational data in XML", see <http://www.soe.ucsc.edu/~wctan/papers/2002/silkroute-TODS.pdf>.
- [17] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene Shekita, Catalina Fan, John Funderburk: "Querying XML Views of Relational Data", VLDB 2001.
- [18] Jayavel Shanmugasundaram, Eugene Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Efstratios Viglas, Jeffery Naughton, Igor Tatarinov: "A General Technique for Querying XML Documents using a relational Database System". See <http://www.cs.cornell.edu/people/jai/papers/GeneralXMLQuerying.pdf>
- [19] Feng.Tian, David.J.DeWitt, Jianjun.Chen and Chun. Zhang. "The design and performance evaluation of alternative XML storage strategies" <http://www.cs.wisc.edu/~czhang/doc/publications/feng6page.pdf>.